How to manipulate curve standards: a white paper for the black hat

Daniel J. Bernstein[1,2], Tung Chou[1], Chitchanok Chuengsatiansup[1], Andreas Hülsing[1], Tanja Lange[1], Ruben Niederhagen[1], and Christine van Vredendaal[1]

[1] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
blueprint@crypto.tw, c.chuengsatiansup@tue.nl, andreas.huelsing@googlemail.com, tanja@hyperelliptic.org, ruben@polycephaly.org, c.v.vredendaal@tue.nl

[2] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to

Abstract. This paper analyzes the cost of breaking ECC under the following assumptions: (1) ECC is using a standardized elliptic curve that was actually chosen by an attacker; (2) the attacker is aware of a vulnerability in some curves that are not publicly known to be vulnerable.

This cost includes the cost of exploiting the vulnerability, but also the initial cost of computing a curve suitable for sabotaging the standard. This initial cost depends upon the acceptability criteria used by the public to decide whether to allow a curve as a standard, and (in most cases) also upon the chance of a curve being vulnerable.

This paper shows the importance of accurately modeling the actual acceptability criteria: i.e., figuring out what the public can be fooled into accepting. For example, this paper shows that plausible models of the "Brainpool acceptability criteria" allow the attacker to target a one-in-a-million vulnerability.

Keywords. Elliptic-curve cryptography, verifiably random curves, verifiably pseudorandom curves, nothing-up-my-sleeve numbers, sabotaging standards, fighting terrorism, protecting the children.

1 Introduction

More and more Internet traffic is encrypted. This poses a threat to our society as it limits the ability of government agencies to monitor Internet communication for the prevention of terrorism and globalized crime. For example, an increasing number of servers use *Transport Layer Security* (TLS) as default (not only for transmissions that contain passwords or payment information) and also most modern chat applications encrypt all communication. This increases the cost of protecting society as it becomes necessary to collect the required information at the end points, i.e., either the servers or the clients. This requires agencies to either convince the service providers to make the demanded information available or to deploy a back door on the client system respectively. Both actions are much more expensive for the agencies than collecting unprotected information from the transmission wire.

Fortunately, under reasonable assumptions, it is feasible for agencies to fool users into deploying cryptographic systems that the users believe are secure but that the agencies are able to break.

1.1 Elliptic-curve cryptography

Elliptic-curve cryptography (ECC) has a reputation for high security and has become increasingly popular. For definiteness we consider the elliptic-curve Diffie–Hellman (ECDH) key-exchange protocol,

This work was supported by the European Commission through the ICT program under contract INFSO-ICT-284833 (PUFFIN), by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005, and by the U.S. National Science Foundation under grant 1018836. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation." Calculations were carried out on two GPU clusters: the Saber cluster at Technische Universiteit Eindhoven; and the K10 cluster at the University of Haifa, funded by ISF grant 1910/12. This work did not receive the funding that it so richly deserves from the U.S. National Security Agency. Permanent ID of this document: bada55ecd325c5bfeaf442a8fd008c54. Date: 2014.07.22.

specifically "ephemeral ECDH", which has a reputation of being the best way to achieve forward secrecy. The literature models ephemeral ECDH as the following protocol $\mathrm{ECDH}_{E,P}$, Diffie–Hellman key exchange using a point P on an elliptic curve E:

1. Alice generates a private integer a and sends aP, the ath multiple of P on E.
2. Bob generates a private integer b and sends bP.
3. Alice computes abP as the ath multiple of bP.
4. Bob computes abP as the bth multiple of aP.
5. Alice and Bob encrypt data using a secret key derived from abP.

There are various published attacks showing that this protocol is breakable for many elliptic curves E, no matter how strong the encryption is. See Section 2 for details. However, there are also many (E, P) for which the public literature does not indicate any security problems. Similar comments apply to, e.g., elliptic-curve signatures.

This model begs the question of where the curve (E, P) comes from. The standard answer is that a central authority generates a curve for the public (while advertising the resulting benefits for security and performance); see, e.g., [25,13,16,7,26,4,1,12]. This does not mean that the public will accept arbitrary curves; our main objective in this paper is to analyze the security consequences of various possibilities for what the public will accept. The general picture is that Alice, Bob, and the central authority Jerry are actually participating in the following three-party protocol ECDH_A, where A is a function determining the public acceptability of a standard curve:

−1. Jerry generates a curve E, a point P, and auxiliary data S such that $A(E, P, S) = 1$.
0. Alice and Bob verify that $A(E, P, S) = 1$.
1. Alice generates a private integer a and sends aP.
2. Bob generates a private integer b and sends bP.
3. Alice computes abP as the ath multiple of bP.
4. Bob computes abP as the bth multiple of aP.
5. Alice and Bob encrypt data using a secret key derived from abP.

Our paper is targeted at Jerry. We make the natural assumption that Jerry is cooperating with a heroic eavesdropper Eve to break the encryption used by potential criminals Alice and Bob. The central question is how Jerry can use his curve-selection flexibility to minimize the attack cost.

Obviously the cost c_A of breaking ECDH_A depends on A, the same way that the cost $c_{E,P}$ of breaking $\mathrm{ECDH}_{E,P}$ depends on (E, P). One might think that, to evaluate c_A, one simply has to check what the public literature says about $c_{E,P}$, and then minimize $c_{E,P}$ over all (E, P, S) with $A(E, P, S) = 1$. The reality is more complicated, for three reasons:

1. There may be vulnerabilities not known to the public: curves E for which $c_{E,P}$ is smaller than indicated by the best public attacks. Our starting assumption is that Jerry and Eve are secretly aware of a vulnerability that applies to a fraction ϵ of all isomorphism classes of curves that the public believes to be secure. The obvious strategy for Jerry is to standardize a vulnerable curve.
2. Some choices of A limit the number of curves E for which there *exists* suitable auxiliary data S. If $1/\epsilon$ is much larger than this limit then Jerry cannot expect any vulnerable (E, P, S) to have $A(E, P, S) = 1$. Fortunately for Jerry, this limit turns out to be much larger than the public thinks it is. See Section 5.
3. Other choices of A do not limit the number of vulnerable E for which S *exists* but nevertheless complicate Jerry's task of *finding* a vulnerable (E, P, S) with $A(E, P, S) = 1$. See Section 4 for analysis of the cost of this computation.

If Jerry succeeds in finding a vulnerable (E, P, S) with $A(E, P, S) = 1$, then Eve simply exploits the vulnerability, obtaining access to the information that Alice and Bob have encrypted for transmission.

Of course, this could require considerable computation for Eve, depending on the details of the secret vulnerability. Obviously, given the risk of this paper being leaked to the public, it would be important for us to avoid discussing details of secret vulnerabilities, even if we were aware of such

vulnerabilities.[3] Our goal in this paper is not to evaluate the cost of Eve's computation, but rather to evaluate the impact of A and ϵ upon the cost of Jerry's computation.

For this evaluation it is adequate to use simplified models of secret vulnerabilities. We specify various artificial curve criteria that have no connection to vulnerabilities but that are satisfied by (E, P, S) with probability ϵ for various sizes of ϵ. We then evaluate how difficult it is for Jerry to find (E, P, S) that satisfy these criteria and that have $A(E, P, S) = 1$.

The reader might object to our specification of ECDH_A as implicitly assuming that the party sabotaging curve choices to protect society is the same as the party issuing curve standards to be used by Alice and Bob. In reality, these two parties are different, and having the first party exercise sufficient control over the second party is often a delicate exercise in finesse. See, for example, [8,6].

1.2 Organization

Section 2 surveys the curve attacks known to the public, including relevant probability analyses. Section 3, as an easy warm-up, shows how to manipulate curve choices when A merely checks for these public vulnerabilities. Section 4 shows how to manipulate "verifiably random" curve choices. Section 5 shows how to manipulate "verifiably pseudorandom" curve choices.

2 Pesky public researchers and their security analyses

One obstacle Jerry has to face in deploying his backdoored elliptic curves is that public researchers have raised awareness of certain weaknesses an elliptic curve may have. Once sufficient awareness of a weakness has been raised, many standardization committees will feel compelled to mention that weakness in their standards. This in turn may alert the targeted users, i.e., the general public: some users will check what standards say regarding the properties that an elliptic curve should have or should not have.

The good thing about standards is that there are so many to choose from. Standards evaluating or claiming the security of various elliptic curves include ANSI X9.62 (1999) [25], IEEE standard P1363 (2000) [13], Certicom SEC 1 v1 (2000) [15], Certicom SEC 2 v1 (2000) [16], NIST FIPS 186-2 (2000) [7], ANSI X9.63 (2001) [26], Brainpool (2005) [4], NSA Suite B (2005) [1], Certicom SEC 1 v2 (2009) [17], Certicom SEC 2 v2 (2010) [18], OSCCA SM2 (2010) [22,23], and ANSSI FRP256V1 (2011) [12]. These standards vary in many details, and also show important variations in public acceptability criteria, an issue explored in depth in subsequent sections of this paper.

Unfortunately, some public criteria have become so widely known that all of the above standards agree upon them. Jerry's curves need to satisfy these criteria. This means not only that Jerry will be unable to use these public attacks as back doors, but also that Jerry will have to bear these criteria in mind when searching for a vulnerable curve. It is possible that the vulnerability known secretly to Jerry does not occur in curves that satisfy the public criteria; on the other hand, it is also possible that this vulnerability occurs *more* frequently in curves that satisfy the public criteria than in general curves. The chance ϵ of a curve being vulnerable is defined relative to the curves that the public will accept.

This section reviews these standard criteria for "secure" curves along with attacks those pesky researchers have found. Jerry must be careful, when designing and justifying his curve, to avoid revealing attacks outside this list; such attacks are not known to the public.

2.1 Review of public ECDLP security criteria

For simplicity we cover only prime fields here. The above standards from the last ten years use only prime fields. If Jerry's secret vulnerability works only in binary fields then we would expect Jerry to have a harder time convincing his targets to use vulnerable curves, although of course he should try.

[3] Note to any journalists who might end up reading this paper: There are no secret vulnerabilities. Really. ECC is perfectly safe. You can quote Jerry.

Let E be an elliptic curve defined over a large prime field $\mathbb{F}_p$. One can always write E in the form $y^2 = x^3 + ax + b$. Most curve standards choose $a = -3$ for efficiency reasons. Practically all curves have low-degree isogenies to curves with $a = -3$, so this choice does not affect security.

Write $|E(\mathbb{F}_p)|$ for the number of points on E defined over $\mathbb{F}_p$. Define ℓ as the largest prime factor of $|E(\mathbb{F}_p)|$, and define the "cofactor" h as $|E(\mathbb{F}_p)|/\ell$. Let P be a point on E of order ℓ.

Elliptic curve cryptography is based on the believed hardness of the *elliptic-curve discrete-logarithm problem* (ECDLP), i.e., the belief that it is computationally infeasible to find a scalar k satisfying $Q = kP$ given a random multiple Q of P on E. The state-of-the-art public algorithm for solving the ECDLP is Pollard's rho method (with negation), which on average requires approximately $0.886\sqrt{\ell}$ point additions. Most publications require the value ℓ to be large; for example, the SafeCurves web page [3] requires that $0.886\sqrt{\ell} > 2^{100}$.

Some standards put upper limits on the cofactor h, but the limits vary. FIPS 186-2 [7, page 24] claims that "for efficiency reasons, it is desirable to take the cofactor to be as small as possible"; the 2000 version of SEC 1 [15, page 17] required $h \leq 4$; but the 2009 version of SEC 1 [17, pages 22 and 78] claims that there are efficiency benefits to "some special curves with cofactor larger than four" and thus requires merely $h \leq 2^{t/8}$ for security level 2^t. We analyze a few possibilities for h and later give examples with $h = 1$; many standard curves have $h = 1$.

Another security parameter is the *complex-multiplication field discriminant* (CM field discriminant) which is defined as $D = (t^2 - 4p)/s^2$ if $(t^2 - 4p)/s^2 \equiv 1 \pmod 4$ or otherwise $D = 4(t^2 - 4p)/s^2$, where t is defined as $p + 1 - |E(\mathbb{F}_p)|$ and s^2 is the largest square dividing $t^2 - 4p$. One standard, Brainpool, requires $|D|$ to be large (by requiring a related quantity, the "class number", to be large). However, other standards do not constrain D, there are various ECC papers choosing curves where D is small, and the only published attacks related to the size of D are some improvements to Pollard's rho method on a few curves. If Jerry needs a curve with small D then it is likely that Jerry can convince the public to accept the curve. We do not pursue this possibility further.

All standards prohibit efficient *additive and multiplicative transfers*. An additive transfer reduces the ECDLP to an easy DLP in the additive group of $\mathbb{F}_p$; this transfer is applicable when ℓ equals p. A degree-k multiplicative transfer reduces the ECDLP to the DLP in the multiplicative group of $\mathbb{F}_{p^k}$ where the problem can be solved efficiently using index calculus if the *embedding degree* k is not too large; this transfer is applicable when ℓ divides $p^k - 1$. All standards prohibit $\ell \in \{p - 1, p, p + 1\}$ and ℓ dividing various larger $p^k - 1$; the exact limit on k varies from one standard to another.

2.2 Review of public ECC security criteria

The most extensive public list of requirements is on the SafeCurves web page [3]. SafeCurves covers hardness of ECDLP, generally imposing more stringent constraints than the standards listed above; for example, SafeCurves requires $|D| > 2^{100}$ and requires the order of p modulo ℓ, i.e., the embedding degree, to be at least $(\ell - 1)/100$. Potentially more troublesome for Jerry is that SafeCurves also covers the general security of ECC, i.e., the security of ECC implementations.

For example, if an implementor of NIST P-224 ECDH uses the side-channel-protected scalar-multiplication algorithm recommended by Brier and Joye [5], reuses an ECDH key for more than a few sessions, and fails to perform a moderately expensive input validation that has no impact on normal usage, then a *twist attack* finds the user's secret key using approximately 2^{58} elliptic-curve additions. See [3] for details. SafeCurves prohibits curves with low *twist security*, such as NIST P-224.

Luckily for Jerry, the other standards listed above focus on ECDLP hardness and impose very few additional ECC security constraints. This gives Jerry the freedom (1) to choose a non-SafeCurves-compliant curve that encourages insecure ECC implementations even if ECDLP is difficult, and (2) to deny that there are any security problems. Useful denial text can be found in a May 2014 presentation [11] from NIST: "The NIST curves do NOT belong to any known class of elliptic curves with weak security properties. No sufficiently large classes of weak curves are known."

Unfortunately, there is some risk that twist-security and other SafeCurves criteria will be added to future standards. This paper considers the possibility that Jerry is forced to generate twist-secure curves; it is important for Jerry to be able to sabotage curve standards even under the harshest

conditions. Obviously it is also preferable for Jerry to choose a curve for which *all* implementations are insecure, rather than merely a curve that encourages insecure implementations.

Twist-security requires the twist E' of the original curve E to be secure. If $|E(\mathbb{F}_p)| = p + 1 - t$ then $|E'(\mathbb{F}_p)| = p + 1 + t$. Define ℓ' as the largest prime factor of $p + 1 + t$. SafeCurves requires $0.886\sqrt{\ell'} > 2^{100}$ to prevent Pollard's rho method; $\ell' \neq p$ to prevent additive transfers; and p having order at least $(\ell' - 1)/100$ modulo ℓ' to prevent multiplicative transfers. SafeCurves also requires various "combined attacks" to be difficult; this is automatic when ℓ is very close to $p+1-t$ and ℓ' is very close to $p + 1 + t$.

2.3 Probability analysis

This subsection analyzes the probability of random curves passing the public criteria described above. The important quantities for Jerry are (1) this probability, (2) the probability ϵ that curves passing the public criteria are secretly vulnerable to Jerry's attack, and (3) the number of curves that Jerry can afford to try and is allowed to try, depending on various optimizations and constraints discussed in subsequent sections. Combining these quantities produces Jerry's overall success chance at creating a vulnerable curve that passes the public criteria, avoiding alarms from the pesky researchers.

Recall that $|E(\mathbb{F}_p)| = p + 1 - t$ and $|E(\mathbb{F}_p)| = h\ell$. Hence, ℓ is a divisor of $p + 1 - t$. Furthermore, Hasse's theorem (see e.g. [21]) states that t is between $-2\sqrt{p}$ and $2\sqrt{p}$.

We begin by analyzing how many random curves have small cofactors. As illustrations we consider cofactor $h = 1$, cofactor $h = 2$, and cofactor $h = 4$. Note that, for primes p large enough to pass a laugh test (at least 224 bits), curves with these cofactors automatically satisfy the requirement $0.886\sqrt{\ell} > 2^{100}$; in other words, requiring a curve to have a small cofactor supersedes requiring a curve to meet minimal public requirements for security against Pollard's rho method.

Let $\pi(x)$ be the number of primes $p \leq x$, and let $\pi(S)$ be the number of primes p in a set S. The prime number theorem states that the ratio between $\pi(x)$ and $x/\log x$ converges to 1 as $x \to \infty$, where log is the natural logarithm. Explicit bounds such as [19] are unfortunately not sufficient to count the number of primes in a short interval $I = [x - y, x]$, but there is nevertheless ample experimental evidence that $\pi(I)$ is very close to $y/\log x$ when y is larger than $\sqrt{x}$.

The number of integers in I of the form $\ell, 2\ell$, or 4ℓ, where ℓ is prime, is the same as the total number of primes in the intervals $I_1 = [x-y, x]$, $I_2 = [(x-y)/2, x/2]$ and $I_4 = [(x-y)/4, x/4]$, namely

$$\pi(I_1 \cup I_2 \cup I_4) \approx \frac{y}{\log x} + \frac{y/2}{\log(x/2)} + \frac{y/4}{\log(x/4)} = \sum_{h=1,2,4} \frac{y/h}{\log(x/h)}. \tag{1}$$

Take $x = p + 1 + 2\sqrt{p}$ and $y = 4\sqrt{p}$ to see that the number of such integers in the Hasse interval is

$$\pi(I_1 \cup I_2 \cup I_4) \approx \sum_{h=1,2,4} \frac{4\sqrt{p}/h}{\log((p+1+2\sqrt{p})/h)}.$$

The total number of integers in the Hasse interval is almost exactly $4\sqrt{p}$, so the chance of an integer in the interval having the form $\ell, 2\ell, 4\ell$ is almost exactly

$$\pi(I_1 \cup I_2 \cup I_4)/(4\sqrt{p}) \approx \sum_{h=1,2,4} \frac{1}{h\log((p+1+2\sqrt{p})/h)}. \tag{2}$$

This does not imply, however, that the same approximation is valid for the number of points on a random elliptic curve. To test the approximation we performed the following experiment: take $p = 2^{224} - 2^{96} + 1$ (the NIST P-224 prime which is also used in following sections), and count the number of points on many random curves. Within a set of 383800 random curves, we found 1430 curves for $h = 1$, found 1117 curves for $h = 2$, and found 934 curves for $h = 4$. The total chance of $\ell, 2\ell, 4\ell$ is approximately 0.00907, whereas Equation (2) evaluates to approximately 0.0113, a gap of about 25%. The ratios between the individual proportions for $h = 1$, $h = 2$, and $h = 4$ are even farther from the ratios in Equation (2): for example, the approximation is about 72% too optimistic for $h = 1$ but about 33% too pessimistic for $h = 4$.

It has been pointed out before that group orders of elliptic curves tend to have small factors more frequently than uniform random integers. See, e.g., the analysis by McKee [10]. However, the correlations between these factors, and the resulting probabilities of finding primes with various cofactors, seem to have been insufficiently analyzed.

We next analyze the probability of meeting the SafeCurves embedding-degree requirement. Luca, Mireles, and Shparlinski gave a theoretical estimate [9] for the probability that for a sufficiently large prime number p and a positive integer K with $\log K = O(\log \log p)$ a randomly chosen elliptic curve $E(\mathbb{F}_p)$ has embedding degree $k \leq K$. This result shows that curves with small embedding degree are very rare, but the SafeCurves bound $K = (\ell - 1)/100$ is not within the bounds of their theorem. Experimentally we found that 3446 curves out of our 3481 random small-cofactor curves had a high embedding degree $\geq K$, and none of them had $\ell = p$.

Random curves have a large CM field discriminant, practically always meeting the SafeCurves CM criterion. None of our experiments found a CM field discriminant below 2^{100}.

The conclusion is that as long as Jerry's vulnerability is not linked to the special properties tested above, his only obstacle to getting his vulnerable curves deployed is that they need to have almost prime order. If he can neglect twist security then about 1 in 111 random curves passes the criteria. This is equivalent to a probability of 0.0090, i.e., about 2^{-7}.

To analyze twist security, we directly conducted an experiment by randomly generating curves counting how many of them pass the security criteria on both the original curve and the twist. We found that only 15 out of 383800 curves had both the curve and its twist having almost prime order and high embedding degree. These numbers imply that with probability roughly 2^{-15} a random curve is protected according to the standard security criteria plus twist security.

3 Manipulating curves

In this section, we target users with minimal acceptability criteria: i.e., we assume that $A(E, P, S)$ checks only the public security criteria for (E, P) described in the previous section. The auxiliary data S might be used to communicate, e.g., a precomputed $|E(\mathbb{F}_p)|$ to be verified by the user, but is not used to constrain the choice of (E, P). Curves that pass the acceptability criteria are safe against known attacks, but have no protection against Jerry's secret vulnerability.

3.1 Curves without public justification

Here are two examples of standard curves distributed without any justification for how they were chosen. These examples suggest that there are many ECC users who do in fact have minimal acceptability criteria.

As a first example, we look at the FRP256V1 standard [12] published in 2011 by the Agence nationale de la sécurité des systèmes d'information (ANSSI). This curve is $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where

$$b = \texttt{0xEE353FCA5428A9300D4ABA754A44C00FDFEC0C9AE4B1A1803075ED967B7BB73F},$$
$$p = \texttt{0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03}.$$

Another example is a curve published by the Office of State Commercial Cryptography Administration (OSCCA) in China along with the SM2 algorithms in 2010 (cf. [23,22]). The curve is of the same form as the ANSSI one with

$$b = \texttt{0x28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93},$$
$$p = \texttt{0xFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF}.$$

Each curve E is also accompanied by a point P. The curves meet the ECDLP requirements reviewed in the previous section. The only further data provided with these curves is data that could also have been computed efficiently by users from the above information. Nothing in the curve documentation suggests any verification that would have further limited the choice of curves.

3.2 The attack

The attack is straightforward. Since the only things that users check are the public security criteria, Jerry can continue generating curves for a fixed p (either randomly or not) that satisfy the public criteria until he gets one that is vulnerable to his secret attack. Alternatively, Jerry can generate curves vulnerable to his secret attack and check them against the public security criteria. Every public attack allows efficient computation of vulnerable curves, so it seems likely that the same will be true for Jerry's secret vulnerability. After finding a vulnerable curve, Jerry simply publishes it.

Of course, Jerry's vulnerability must not depend on any properties excluded by the public security criteria, and there must be enough vulnerable curves. Enumerating 2^7 vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient if Jerry can ignore twist-security, and enumerating 2^{15} vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient even if twist-security is required.

Even if Jerry's curves are less frequent, Jerry can increase his chances by also varying the prime p. The standard security requirements do not seriously scrutinize the choice of prime. Any allowed variability in p would also improve the attack in Section 5, but to simplify our analysis we now make the worst-case assumption that Jerry is forced to reuse particular standard primes such as the NIST primes.

3.3 Implementation

We implemented the attack to demonstrate that it is really feasible in practice. In our implementation the same setting as above is adopted and even made more restrictive: the resulting curve should be of the form $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where p is the same as for the ANSSI curve. The public security criteria we consider are all the standard ECDLP security criteria plus twist security, and we further require that both cofactors are 1.

Of course, as explained in the introduction, we will not include any actual secret vulnerability in this white paper. We instead use a highly structured parameter b as an artificial model of a secret vulnerability. We show that we can construct a curve with very structured parameter b that passes all the public criteria.

Our attack is implemented using the Sage computer algebra system [24]. We took

$$b = \texttt{0x5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55EC}$$

as start value and incremented b until we found a curve that meets the public security criteria. This corresponds to Jerry iteratively checking whether curves that are vulnerable to the secret attack fulfill the public criteria.

As a result we found a desired curve, which we call BADA55-R-256, with

$$b = \texttt{0x5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA5A57}$$

after 1131 increments within 78 minutes on a single core AMD CPU.[4] One can easily check using any computer algebra system that the curve does meet all the public criteria. It is thus clear that users who only verify public security criteria can be very easily attacked, and Jerry has an easy time if he is working for or is trusted by ANSSI, OSCCA, or a similar organization.

4 Manipulating seeds

For some audiences verifying public security criteria is not enough; they might demand justifications for the curve choices. In this section, we consider users who are suspicious that the curve parameters might be maliciously chosen to enable a secret attack, as in the previous section. Empirically many

[4] Note that a *lucky* attacker starting from `0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA` is able to find the following secure parameter already within 43 minutes after only 622 increments:

$$b = \texttt{0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48}.$$

users are satisfied if they get a *hash verification routine* as justification like for example in case of ANSI X9.62 [25], IEEE P1363 [13], SEC 2 [18], or NIST FIPS 186-2 [7]. Hash verification routines mean that Jerry cannot use a very small set of vulnerable curves, but we will show below that he has good chances to get vulnerable curves deployed if they are just somewhat more common.

4.1 Hash verification routine

As the name implies a hash verification routine involves a cryptographic hash function. The inputs to the routine are the curve parameters and a seed that is published along with the curve. Usually the seed is hashed to compute a curve parameter or point coordinate. The ways of computing the parameters differ but the public justification is that these bind the curve parameters to the hash value, making them hard to manipulate since the hash function is preimage resistant[5]. In addition the user verifies a set of public security criteria. We focus on the obstacle that Jerry faces and call curves that can be verified with such routines *verifiably hashed curves*.

For Jerry's marketing we do not recommend the phrase "verifiably hashed": it is better to claim that the curves are totally random (even though this is not what is being verified) and that these curves could not possibly be manipulated (even though Jerry is in fact quite free to manipulate them). For example, ANSI X9.62 [25, page 31] speaks of "selecting an elliptic curve verifiably at random"; SEC 2 [18, copy and paste: page 8 and page 18] claims that "verifiably random parameters offer some additional conservative features" and "that the parameters cannot be predetermined". NIST's marketing in [7] is not as good: NIST uses the term "pseudo-random curves".

Below we recall the curve verification routine for the NIST P-curves. The routine is specified in NIST FIPS 186-2 [7] and is based on ANSI X9.62 and IEEE P1363 [13].

Each NIST P-curve is of the form $y^2 = x^3 - 3x + b$ over a prime field $\mathbb{F}_p$ and is published with a seed s. The hash function SHA-1 is denoted as `SHA1`; recall that SHA-1 produces a 160-bit hash value. The bit length of p is denoted by m. We use $\text{bin}(i)$ to denote the binary expansion of some integer i and use $\text{int}(j)$ to denote the integer with binary expansion j. For given parameters b, p, and s, the verification routine is:

1. Let $z \leftarrow \text{int}(s)$. Compute $h_i \leftarrow \texttt{SHA1}(s_i)$ for $0 \le i \le v$, where $s_i \leftarrow \text{bin}((z+i) \bmod 2^{160})$ and $v = \lfloor (m-1)/160 \rfloor$.
2. Let h be the rightmost $m-1$ bits of $h_0 || h_1 || \cdots || h_v$. Let $c \leftarrow \text{int}(h)$.
3. Verify that $b^2 c = -27$ in $\mathbb{F}_p$.

To generate a verifiably hashed curve one starts with a seed and then follows the same steps 1 and 2 as above. Instead of step 3 one tries to solve for b given c; this succeeds in about 50% of all choices for s. The public perception is that this process is repeated with fresh seeds until the resulting curve satisfies all public security criteria.

4.2 Acceptability criteria

One might think that the public acceptability criteria are defined by the P1363/NIST verification routine stated above: i.e., $A(E, P, s) = 1$ if and only if (E, P) passes the public security criteria from Section 2 and (E, s) passes the verification routine stated above with seed s and E defined as $y^2 = x^3 - 3x + b$.

However, the public acceptability criteria are not actually so strict. P1363 allows $y^2 = x^3 + ax + b$ without the requirement $a = -3$. P1363 does require $b^2 c = a^3$ where c is a hash as above, but neither P1363 nor NIST gives a justification for the relation $b^2 c = a^3$, and it is clear that the public will accept different relations. For example, the Brainpool curves (see Section 5) use the simpler relations $a = g$ and $b = h$ where g and h are separate hashes. One can equivalently view the Brainpool curves as following the P1363 procedure but using a different hash for c, namely computing c as g^3/h^2 where again g and h are separate hashes. Furthermore, even though NIST and Brainpool both use SHA-1,

[5] If Jerry has a back door in the hash function this situation is no different than in the previous section, so we will not assume this feature.

SHA-1 is not the only acceptable hash function; for example, Jerry can easily argue that SHA-1 is outdated and should be replaced by SHA-2 or SHA-3.

We do not claim that the public would accept *any* relation, or that the public would accept *any* choice of "hash function", allowing Jerry just as much freedom as in Section 3. The exact boundaries of public acceptability are complicated and not immediately obvious. We have carried out small-scale unscientific surveys to determine approximations to these boundaries (see Section 5), and we encourage Jerry to carry out larger-scale surveys, while taking care to prevent leaks to the public.

4.3 The attack

Jerry begins the attack by defining a public hash verification routine. As explained above, Jerry has some flexibility to modify this routine. This flexibility is not *necessary* for the rest of the attack in this section (for example, Jerry can use exactly the NIST verification routine) but a more favorable routine does improve the *efficiency* of the attack. Our cost analysis below makes a particularly efficient choice of routine.

Jerry then tries one seed after another until finding a seed for which the verifiably hashed curve (1) passes the public security criteria but (2) is subject to his secret vulnerability. Jerry publishes this seed and the resulting curve, pretending that the seed was the first random seed that passed the public security criteria.

4.4 Optimizing the attack

Assume that the curves vulnerable to Jerry's secret attack are randomly distributed over the curves satisfying the public security criteria. Then the success probability that a seed leads to a suitable curve is the probability that a curve is vulnerable to the secret attack times the probability that a curve satisfies the public security criteria. Depending on which condition is easier to check Jerry runs many hash computations to compute candidate bs, checks them for the easier criterion and only checks the surviving choices for the other criterion. The hash computations and security checks for each seed are independent from other seeds; thus, this procedure can be parallelized with an arbitrary number of parallel computing instances.

We generated a family of curves to show the power of this method and highlight the computing power of hardware accelerators (such as GPUs or Xeon Phis). We began by defining our own curve verification routine and implementing the corresponding secret generation routine. The hash function we use is Keccak with 256-bit output instead of SHA-1. The hash value is $c = \text{int}(\text{Keccak}(s))$, and the relation is simply $b = c$ in $\mathbb{F}_p$. All choices are easily justified: Keccak is the winner of the SHA-3 competition and much more secure than SHA-1; using a hash function with a long output removes the weird order of hashed components that smells suspicious and similarly $b = c$ is as simple and unsuspicious as it can get. In reality, however, these choices greatly benefit the attack: the GPUs efficiently search through many seeds in parallel, one single computation of Keccak has a much easier data flow than in the method above, and having b computed without any expensive number-theoretic computation (such as square roots) means that the curve can be tested already on the GPUs and only the fraction that satisfies the first test is passed on to the next stage. Of course, for a real vulnerability we would have to add the cost of checking for that vulnerability, but minimizing overhead is still useful.

Except for the differences stated above, we followed closely the setting of the NIST P-curves. The target is to generate curves of the form $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, and we consider 3 choices of p: the NIST P-224, P-256, and P-384 primes. (For P-384 we switched to Keccak with 384-bit output.) As a placeholder "vulnerability" we define E to be vulnerable if b starts with the hex-string `BADA55EC`. This fixes 8 hex digits, i.e., it simulates a 1-out-of-2^{32} attack. In addition we require that the curves meet the standard ECDLP criteria plus twist security and have both cofactors equal to 1.

4.5 Implementation

Our implementation uses NVIDIA's CUDA framework for parallel programming on GPUs. A high-end GPU today allows several thousands of threads to run in parallel, though at a frequency slightly lower

than high-end CPUs. We let each thread start with its own random seed. The threads then hash the seeds in parallel. After hashing, each thread outputs the hash value if it starts with the hex-string BADA55EC. To restart, each seed is simply increased by 1 such that no new source of randomness is required. Checking whether outputs from GPUs also satisfy the public security criteria is done by running a Sage [24] script on CPUs. Since only 1 out of 2^{32} curves has the desired pattern, the CPU computation is totally hidden by GPU computation. Longer strings, corresponding to less likely vulnerabilities, make GPUs even more powerful for our attack scheme. Note that this situation has changed over the last 15 years; at the time that the NIST curves were generated, their procedure would have been close to optimal for an attack performed on the available CPU architectures, especially with SHA-1 implemented in hardware.

In the end we found 3 "vulnerable" verifiably hashed curves: BADA55-VR-224, BADA55-VR-256, and BADA55-VR-384, each corresponding to one of the three NIST P-curves. Curve parameters can be found on the BADA55 web page [2] and in Appendix A.

As an example, BADA55-VR-256 was found within 7 hours, using a cluster of 41 NVIDIA GTX780 GPUs (http://blog.cr.yp.to/20140602-saber.html). Each GPU is able to carry out 170 million 256-bit-output Keccak hashes in a second. Most of the instructions are bitwise logic instructions. On average each core performs 0.58 bitwise logic instructions per cycle while the theoretical maximum throughput is 0.83. We have two explanations for the gap: first, each thread uses many registers, which makes the number of active warps too small to fully hide the instruction latency; second, there is not quite enough instruction-level parallelism to fully utilize the cores in this GPU architecture. We also tested our implementation on K10 GPUs. Each of them carries out only 61 million hashes per second. This is likely to be caused by register spilling: the K10 GPUs have only 63 registers per thread instead of the 255 registers of the GTX780. Using a sufficient amount of computing power easily allows Jerry to deal with secret vulnerabilities that have smaller probabilities of occurrence than 2^{-32}.

5 Manipulating nothing-up-my-sleeve numbers

There are some particularly pesky researchers who do not shut up even when provided with a verification routine as in the previous section. These researchers might even think of the powerful attack presented in the previous section. We will now highlight two such cases and then show how Jerry can deal with such overly sceptic people. In 1999, M. Scott wrote as a reaction to the announcement of the NIST curves [20]:

> [...] Consider now the possibility that one in a million of all curves have an exploitable structure that "they" know about, but we don't. Then "they" simply generate a million random seeds until they find one that generates one of "their" curves. Then they get us to use them. And remember the standard paranoia assumptions apply - "they" have computing power way beyond what we can muster. So maybe that could be 1 billion.
> A much simpler approach would generate more trust. Simply select B as an integer formed from the maximum number of digits of pi that provide a number B which is less that p. Then keep incrementing B until the number of points on the curve is prime. Such a curve will be accepted as "random" as all would accept that the decimal digits of pi have no unfortunate interaction with elliptic curves. We would all accept that such a curve had not been specially "cooked".
> So, sigh, why didn't they do it that way? Do they want to be distrusted?

In the same vein the German ECC Brainpool [4] consortium raised scepticism and suggested using natural constants in place of random seeds. They coined the term "verifiably pseudorandom" for this method of generating seeds. Others speak of "nothing-up-my-sleeves numbers", a nice reference to magicians which we will take as an inspiration to our endeavor to show how Jerry can play this system. We comment that "nothing-up-my-sleeves numbers" also appear in other areas of cryptography and can be manipulated in similar ways, but this paper focuses on manipulation of elliptic curves.

5.1 Public procedure

We first present the procedure that Brainpool is using to produce curve parameters [4, Section 5.2]:

1. Obtain a 160-bit seed s from the hexadecimal representation of e.
2. Compute $h \leftarrow \texttt{SHA1}(s)$.
3. Deterministically expand h to the required length for the curve parameter a.
4. Let $a \leftarrow \text{int}(h)$ and check if $-3 = a \cdot u^4$ is solvable in $\mathbb{F}_p$. If so, compute and store u. Otherwise update the seed s to $s \leftarrow (s+1) \bmod 2^{160}$ and return to step 2.
5. Set $\text{seed}_a \leftarrow s$ and update $s \leftarrow (s+1) \bmod 2^{160}$.
6. Compute $h \leftarrow \texttt{SHA1}(s)$.
7. Deterministically expand h to the required length for the curve parameter b; let $b \leftarrow \text{int}(h)$.
8. Check that the elliptic curve E over $\mathbb{F}_p$ given by $y^3 = x^3 + ax + b$ fulfills all required security features. If not, update the seed s to $s \leftarrow (s+1) \bmod 2^{160}$ and return to step 2.
9. Set $\text{seed}_b \leftarrow s$ and update $s \leftarrow (s+1) \bmod 2^{160}$.
10. Compute $h \leftarrow \texttt{SHA1}(s)$.
11. Deterministically expand h; let $t \leftarrow \text{int}(h)$.
12. Find the base point P_0 with the smallest x-coordinate. Compute $P \leftarrow tP_0$.
13. Set $\text{seed}_P \leftarrow s$.
14. Return the curve parameters a, b, P, and u and the seeds seed_a, seed_b, and seed_P.

Despite the apparent rigidity coming from the natural constant e and the deterministic nature of this algorithm, there are actually several arbitrary choices that Jerry can use in his favor as we show in the following.

5.2 The attack

In order to find a well-defined, deterministic procedure that leads to a curve with the desired vulnerability, Jerry needs a sufficiently large set of plausible choices for the generation algorithm that allows Jerry to pinpoint a specific choice resulting in appropriate curve parameters. When defining the generation procedure, Jerry is free, e.g., to choose a particular hash function, the length of the input seed, the update function between seeds, and the initial constant for deriving the seed. The basic idea for the attack is to try all possible combinations of hash functions, seed length, initial seed constant, and so on, until he finds a curve that exhibits the desired vulnerability.

Choice of hash function. There are many easily justified choices for the hash function. Following Brainpool, Jerry could use SHA-1; since SHA-1 is considered less secure obvious replacements are SHA-256 or SHA-512. Yet another possibility is to use Keccak, the recent SHA-3 competition winner. Keccak offers many variations with different initialization vectors and padding rules; the security level of Keccak is adjustable by choosing several different options for r (bit rate) and c (capacity). Examples of the Keccak family are Keccak-224, Keccak-256, Keccak-384, Keccak-512, "default" Keccak (capacity $c = 576$ with 128 bytes of output), Keccak-128 (capacity $c = 256$ with 168 bytes of output), SHA3-224, SHA3-256, SHA3-384, and SHA3-512. All of these Keccak/SHA-3 choices can be implemented efficiently with a single code base and variable input parameters. For our example the randomness gained from the Keccak choices was sufficient so that we did not implement other hash functions.

Seed length. The Brainpool standard uses a seed length of 160 bits. There is no explanation why the length of 160-bit was chosen. Some general remarks refer to previous standards which also use 160-bit seeds. Obviously, there are more plausible possibilities for the seed length. Jerry can easily argue for any seed length that is longer than 160 bits, e.g., to use 256 bits since it provides "cryptographic parity"[14] with SHA-256, or to use 512 bits because it provides very high security. A shorter seed length may be harder to sell to a sceptical audience.

Update function. Brainpool uses an increment modulo 2^{160} on the seed in order to step to valid, secure curve parameters. There are more options for the implementation of the seed update, e.g., using a 32-bit counter at the beginning or at the end of the seed, interpreting the bit-sequence of the seed as an integer in big-endian or little-endian order, and so on. Using a counter is preferable for parallel implementations because cores can search different counter regions separately within a range that is likely to contain at most one curve satisfying the public security criteria. Once a good candidate is found, Jerry needs to check that this would be the first suitable curve for the deterministic increments.

Choice of constant. Brainpool uses e, the base of natural logarithms; there are no reasons given why this particular value was chosen. Another possibility is π, used by Brainpool for the verifiably pseudorandom generation of primes. These values can be used with the justification to follow the Brainpool standard.

More options can be derived by following different standards that use different values. For example, $\sqrt{2}$ is the first constant used in SHA-1 and $\sqrt[3]{2}$ is the first round constant used in SHA-2. The golden ratio $(1+\sqrt{5})/2$ is used in RC5 and the Tiny Encryption Algorithm, $\sin(1)$ is the first constant used in MD5, and $1/\pi$ is the constant used in ARIA. More possibilities are sin, cos, tan, cosh, ..., $\sqrt{x}$, $\sqrt[3]{x}$, ... of small integers or small prime numbers, their reciprocals, and using other transcendental numbers, e.g., the Riemann zeta function at odd integers.

Jerry could be creative and use previously unused numbers such as numbers derived from some historical document or newspaper, personal information of, e.g., arbitrary celebrities in an arbitrary order, arbitrary collections of natural or physical constants and even a combination of several sources. If the public accepts numbers with such flimsy justifications as "nothing-up-my-sleeves numbers" then Jerry obviously has as much flexibility as in Section 4. We make the worst-case assumption that the public is not quite as easily fooled, and similarly that the public would not accept $703e^{(\sqrt[8]{30}+4\pi)/9\sin(\sqrt[3]{16})}$ as a "nothing-up-my-sleeve number".

5.3 Implementation

For this example, we are aiming at finding a plausible procedure that generates a verifiably pseudorandom curve. As "vulnerability" we use that the string `BADA55` appears within the hexadecimal representation of the curve parameter a. We use the NIST prime P-224, thus there are 200 possible starting positions for the `BADA55` string, meaning that this simulates a vulnerability with probability $200/2^{24} \sim 2^{-16}$.

We begin with 17 natural constants: π, e, Euler gamma, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{7}$, $\log(2)$, $(1+\sqrt{5})/2$, $\zeta(3)$, $\zeta(5)$, $\sin(1)$, $\sin(2)$, $\cos(1)$, $\cos(2)$, $\tan(1)$, and $\tan(2)$. We extend this list by including reciprocals. We then convert to constants between 0 and 1 in three different ways: take the fractional part; divide by 256 and discard any minus sign; or take all bits starting from the most significant (i.e., divide by an appropriate power of 2), again discarding any minus sign. Removing duplicates produces 73 starting seeds.

For the hash function, we are using the 10 variants of Keccak listed above. As length for each seed, we use either 20, 32, 48, 64, or 128 bytes, or the block size of the Keccak configuration (6 choices). The seed is either stored in big-endian or little-endian format (2 choices). In order to update the seed during the generation procedure, we are using a counter of length either 0 (i.e., the seed itself is incremented), 2, 3, 4, or 8 bytes (5 choices), either at the beginning or at the end of the seed (2 choices). We either truncate the seed to the right length or round it to the right length (2 choices); note that in many cases these are identical. Finally, we are using several different ways to update the counter between generation of a and b and to choose the order of generating a and b (8 choices). All in all, we have $73 \cdot 10 \cdot 6 \cdot 2 \cdot 5 \cdot 2 \cdot 2 \cdot 8 = 1401600$ possible configurations, mostly different, with a high probability to find a procedure that produces the desired vulnerability.

To make the verification procedure seemingly deterministic, we have to use the first curve parameters (using the specified update function) that result in a "secure" curve according to the public security tests. However, performing all public security tests for each iteration candidate during the attack is very costly. Instead, we split the attack into two steps:

1. For a given verification procedure f_i we iterate over the seeds $s_{i,k}$ using the specific update function of f_i. We check each parameter candidate from seed $s_{i,k}$ for our secret `BADA55` vulnerability. After a certain number of update steps the probability that we passed valid, secure parameters is very high; thus, we discard the verification procedure and start over with another one. If we find a candidate exhibiting the vulnerability, we perform the public security tests on this particular candidate. If the `BADA55` candidate passes, we proceed to step 2.
2. We perform the whole public verification procedure f_i stating with seed $s_{i,0}$ and check whether there is any valid parameter set passing the public security checks already before the `BADA55` parameters are reached. If there is such an earlier parameter set, we return to step 1 with the next verification procedure f_{i+1}.

The largest workload in our attack scenario is step 2, the re-checking for earlier safe curve parameters before `BADA55` candidates. The public security tests are not well suited for GPU parallelization; the first step of the attack procedure is relatively cheap and a GPU parallelization of this step does not have a remarkable impact on the overall runtime. Therefore, we implemented the whole attack only for the CPUs of the Saber cluster and left the GPUs idle. We chose 8000 as the limit for the update counter to have a good chance that the first twist-secure curve starting from the seed is the curve with our vulnerability. This means that in total $1401600 \cdot 8000 \sim 2^{33}$ curves are covered of which we expect 2^{18} to be twist secure with small cofactor.

5.4 Example

Using the above described freedom in the specification of the curve generation procedure, we computed (using a few hours on a cluster) the following verification procedure which outputs a curve with `BADA55` in a. In fact we easily found several examples and here show the nicest one: the counter was increased only about a hundred times, the `BADA55` string appears very early in a, and the starting value is very clearly within the boundaries of public acceptability. We could have easily chosen a more restrictive "vulnerability".

This is how Jerry can present this procedure:

BADA55-VPR-224 uses a verifiably pseudorandom curve-generation procedure for maximum security. The natural starting value is cos(1) in order to make BADA55-VPR-224 independent of previous designs and to increase the design diversity: Brainpool already uses $\exp(1) = e$ and $\arctan(1) = \pi/4$ (π and $\pi/4$ are equivalent here since bits are taken starting from the most significant), and MD5 already uses sin(1). BADA55-VPR-224 generates the full 160-bit seed as a 32-bit counter followed by cos(1). The seed update to generate b after generating a is to complement the seed; this ensures maximal difference between these parameters. BADA55-VPR-224 uses the state-of-the-art SHA-3 hash function and the widespread NIST prime P-224, i.e., $p = 2^{224} - 2^{96} + 1$.

The generation procedure is as follows; it uses the first counter value that produces a secure curve:

1. Obtain a 160-bit seed s by appending the hexadecimal representation of cos(1) to a 32-bit counter, i.e., $s \leftarrow$ `0x00000000 | 0x8A51407DA8345C91C2466D976871BD2A`.
2. Compute $h \leftarrow$ `SHA3512`(s).
3. Let $a \leftarrow \mathrm{int}(h)$ and set $\mathrm{seed}_a \leftarrow s$.
4. Let t be the bitwise complement of s.
5. Compute $h \leftarrow$ `SHA3512`(t).
6. Let $b \leftarrow \mathrm{int}(h)$.
7. Check that the elliptic curve E over $\mathbb{F}_p$ given by $y^3 = x^3 + ax + b$ fulfills all public security requirements. If not, update the seed s by increasing the counter and return to step 2.
8. Set $\mathrm{seed}_b \leftarrow t$.
9. Return the curve parameters a and b and the seeds seed_a and seed_b for verification.

The first counter value that gives a secure curve is 184 (hex `0xB8`), giving the seeds

$$\mathrm{seed}_a = \texttt{0x000000B88A51407DA8345C91C2466D976871BD2A},$$
$$\mathrm{seed}_b = \texttt{0xFFFFFF4775AEBF8257CBA36E3DB99268978E42D5}.$$

The curve parameters are

$$a = \texttt{0x7144BA12CE8A0C3BEFA053EDBADA555A42391AC64F052376E041C7D4AF23195E}$$
$$\texttt{BD8D83625321D452E8A0C3BB0A048A26115704E45DCEB346A9F4BD9741D14D49},$$
$$b = \texttt{0x5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276B}$$
$$\texttt{A3669AFF6FFC0F4C6AE4AE2E5D74C3C0AF97DCE17147688DDA89E734B56944A2}.$$

6 Manipulating committees

The previous sections already pointed out several marketing tricks Jerry can use to sell his curves. In addition it helps if he can rely on some additional parties:

- Dr. Dan, to advertise properties of Jerry's curve generation routine and spread FUD about other choices;
- Margaret, to propose the curve for Internet protocols;
- Scott, to implement the arithmetic (preferably in hardware) meaning that Jerry can point to a company user who is already committed to Jerry's proposal so that it cannot be changed anymore;

and of course on his position as a member of a trusted agency.

However, there remain procedures for curve standardization that are hard to overcome: Curve25519 is an example of a procedure to derive secure curve parameters that exhibits only a small attack surface for the injection of secret vulnerabilities. Future research will target such overly sceptical curve generation procedures in order to hopefully fool even the most sceptical and security fanatic audience.

References

1. National Security Agency. Suite B cryptography / cryptographic interoperability, 2005. http://www.nsa.gov/ia/programs/suiteb_cryptography/.
2. Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. BADA55 elliptic curves. http://safecurves.cr.yp.to/bada55.html (accessed 1 July 2014).
3. Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to (accessed 1 July 2014).
4. ECC Brainpool. ECC Brainpool standard curves and curve generation, 2005. http://www.ecc-brainpool.org/download/Domain-parameters.pdf.
5. Eric Brier and Marc Joye. Weierstraß elliptic curves and side-channel attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer, 2002.
6. Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Shacham. On the practical exploitability of Dual EC in TLS implementations. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, 2014. USENIX Association. https://projectbullrun.org/dual-ec/index.html.
7. National Institute for Standards and Technology. Digital signature standard, 2000. Federal Information Processing Standards Publication 186-2 http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf.
8. John Kelsey. Choosing a DRBG algorithm. https://github.com/matthewdgreen/nistfoia/blob/master/011%20-%209.12%20Choosing%20a%20DRBG%20Algorithm.pdf.
9. Florian Luca, David Jose Mireles, and Igor E. Shparlinski. MOV attack in various subgroups on elliptic curves. *Illinois Journal of Mathematics*, 48(3):1041–1052, 07 2004.
10. James McKee. Subtleties in the distribution of the numbers of points on elliptic curves over a finite prime field. *Journal of the London Mathematical Society*, 59:448–460, 1999. http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=20335.
11. Dustin Moody. Development of FIPS 186: Digital signatures (and elliptic curves), 2014. http://csrc.nist.gov/groups/ST/crypto-review/documents/FIPS_186_and_Elliptic_Curves_052914.pdf.
12. Agence nationale de la sécurité des systèmes d'information. Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française, 2011. http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/publication-d-un-parametrage-de-courbe-elliptique-visant-des-applications-de.html.
13. Institute of Electrical and Electronics Engineers. IEEE 1363-2000: Standard specifications for public key cryptography, 2000. Preliminary draft at http://grouper.ieee.org/groups/1363/P1363/draft.html.
14. Eric Rescorla and Margaret Salter. Extended random values for TLS, March 2009. Internet-Draft version 02.
15. Certicom Research. SEC 1: Elliptic curve cryptography, version 1.0, 2000. http://www.secg.org/download/aid-385/sec1_final.pdf.

16. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, version 1.0, 2000. http://www.secg.org/download/aid-386/sec2_final.pdf.
17. Certicom Research. SEC 1: Elliptic curve cryptography, version 2.0, 2009. http://www.secg.org/download/aid-780/sec1-v2.pdf.
18. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, version 2.0, 2010. http://www.secg.org/download/aid-784/sec2-v2.pdf.
19. J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.
20. Michael Scott. Re: NIST announces set of Elliptic Curves, 1999. https://groups.google.com/forum/message/raw?msg=sci.crypt/mFMukSsORmI/FpbHDQ6hM_MJ.
21. Josef H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics 106. Springer-Verlag, 2009.
22. China State Commercial Cryptography Administration (OSCCA). Public key cryptographic algorithm SM2 based on elliptic curves, December 2010. http://www.oscca.gov.cn/UpFile/2010122214822692.pdf.
23. China State Commercial Cryptography Administration (OSCCA). Recommanded curve parameters for public key cryptographic algorithm SM2 based on elliptic curves, December 2010. http://www.oscca.gov.cn/UpFile/2010122214836668.pdf.
24. W. A. Stein et al. *Sage Mathematics Software (Version 6.1.1)*. The Sage Development Team, 2014. http://www.sagemath.org.
25. Accredited Standards Committee X9. American national standard X9.62-1999 public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA), 1999. Preliminary draft at http://grouper.ieee.org/groups/1363/Research/Other.html.
26. Accredited Standards Committee X9. American national standard X9.63-2001, public key cryptography for the financial services industry: key agreement and key transport using elliptic curve cryptography, 2001. Preliminary draft at http://grouper.ieee.org/group/1363/Research/Other.html.

A The BADA55 curves

This section presents the "BADA55 curves" mentioned in Section 3, Section 4 and Section 5. Each BADA55 curve is of the form $y^2 = x^3 + ax + b$ over a prime field $\mathbb{F}_p$, where p is one of the NIST or the ANSSI primes. For the BADA55-R curve and the BADA55-VR curves $a = -3$. The curves meet all the SafeCurves criteria, and the cofactors of the curve and its twist are 1 (cf. Section 2). For each curve we present the parameters directly in the Sage [24] script that is used to check the criteria and cofactors.

Listing A.1 The **BADA55-R-256** curve is defined modulo the ANSSI prime $p =$ 0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03.

```
B = 0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48
p = 0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03
k = GF(p)
E = EllipticCurve([-3,k(B)])
n = E.cardinality()
print n != p
print n.is_prime()
print (2*p+2-n).is_prime()
print Integers(n)(p).multiplicative_order() * 100 >= n-1
print Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1
cmdisc = ((p+1-n)^2-4*p).squarefree_part()
if cmdisc % 4 != 1: cmdisc *= 4
print -cmdisc > 2^100
```

Listing A.2 The **BADA55-VR-224** curve is defined modulo the NIST P-224 prime $p = 2^{224} - 2^{96} + 1$. The curve parameter b is generated by hashing the seed s below using Keccak with 256-bit output as for BADA55-VR-256. Since the output is more than 224 bits, we implicitly reduce it modulo p.

```
S = 0x3CC520E9434349DF680A8F4BCADDA648D693B2907B216EE55CB4853DB68F9165
B = 0xBADA55ECFD9CA54C0738B8A6FB8CF4CCF84E916D83D6DA1B78B622351E11AB4E
# not verified in this script: B is keccakc512 hash of S
p = 2^224 - 2^96 + 1
k = GF(p)
E = EllipticCurve([-3,k(B)])
n = E.cardinality()
print n != p
print n.is_prime()
print (2*p+2-n).is_prime()
print Integers(n)(p).multiplicative_order() * 100 >= n-1
print Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1
cmdisc = ((p+1-n)^2-4*p).squarefree_part()
if cmdisc % 4 != 1: cmdisc *= 4
print -cmdisc > 2^100
```

Listing A.3 The **BADA55-VR-256** curve is defined modulo the NIST P-256 prime $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. The curve parameter b is generated by hashing the seed s below using Keccak with 256-bit output, i.e., Keccak with capacity 512.

```
S = 0x3ADCC48E36F1D1926701417F101A75F000118A739D4686E77278325A825AA3C6
B = 0xBADA55ECD8BBEAD3ADD6C534F92197DEB47FCEB9BE7E0E702A8D1DD56B5D0B0C
# not verified in this script: B is keccakc512 hash of S
p = 2^256 - 2^224 + 2^192 + 2^96 - 1
k = GF(p)
E = EllipticCurve([-3,k(B)])
n = E.cardinality()
print n != p
print n.is_prime()
print (2*p+2-n).is_prime()
print Integers(n)(p).multiplicative_order() * 100 >= n-1
print Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1
cmdisc = ((p+1-n)^2-4*p).squarefree_part()
if cmdisc % 4 != 1: cmdisc *= 4
print -cmdisc > 2^100
```

Listing A.4 The **BADA55-VR-384** curve is defined over the NIST P-384 prime $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$. The curve parameter b is generated by hashing the seed s below using Keccak with 384-bit output, i.e., Keccak with capacity 768.

```
S = 0xCA9EBD338A9EE0E6862FD329062ABC06A793575A1C744F0EC24503A525F5D06E
B = 0xBADA55EC3BE2AD1F9EEEA5881ECF95BBF3AC392526F01D4CD13E684C63A17CC4
    D5F271642AD83899113817A61006413D
# not verified in this script: B is keccakc768 hash of S
p = 2^384 - 2^128 - 2^96 + 2^32 - 1
k = GF(p)
E = EllipticCurve([-3,k(B)])
n = E.cardinality()
print n != p
print n.is_prime()
print (2*p+2-n).is_prime()
print Integers(n)(p).multiplicative_order() * 100 >= n-1
print Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1
cmdisc = ((p+1-n)^2-4*p).squarefree_part()
if cmdisc % 4 != 1: cmdisc *= 4
print -cmdisc > 2^100
```

Listing A.5 The **BADA55-VPR-224** curve is defined over the NIST P-224 prime $p = 2^{224} - 2^{96} + 1$. The hash function we use is SHA-3-512. The seed s is composed of the first digits of cos(1) plus a counter; t is the binary complement of s. We then hash s to obtain the curve parameter a and hash t to obtain b. BADA55-VPR-224 is the curve with the smallest counter that satisfies the public security criteria.

```
cos1 = 0x8A51407DA8345C91C2466D976871BD2A
print cos1 == Integer(RealField(128)(cos(1))*2^128)
S = 0x000000B88A51407DA8345C91C2466D976871BD2A
print S == 184*2^128 + cos1
# not verified by this script: 184 is first counter giving secure curve
T = 0xFFFFFF4775AEBF8257CBA36E3DB99268978E42D5
print S+T == 2^160-1
A = 0x7144BA12CE8A0C3BEFA053EDBADA555A42391FC64F052376E041C7D4AF23195E
    BD8D83625321D452E8A0C3BB0A048A26115704E45DCEB346A9F4BD9741D14D49
# not verified in this script: A is hash of S
B = 0x5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276B
    A3669AFF6FFC0F4C6AE4AE2E5D74C3C0AF97DCE17147688DDA89E734B56944A2
# not verified in this script: B is hash of T
p = 2^224 - 2^96 + 1
k = GF(p)
E = EllipticCurve([k(A),k(B)])
n = E.cardinality()
print n != p
print n.is_prime()
print (2*p+2-n).is_prime()
print Integers(n)(p).multiplicative_order() * 100 >= n-1
print Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1
cmdisc = ((p+1-n)^2-4*p).squarefree_part()
if cmdisc % 4 != 1: cmdisc *= 4
print -cmdisc > 2^100
```